GSOC'25 PROPOSAL - OpenWISP

Improve netjsongraph.js resiliency and visualization

Organization: OpenWISP

Project Name: NetJSON

Candidate Name: Yashaswi Kumar

OpenWisp Project Description: Here

Mentors: Nitesh Sinha , Federico Capoano

Expected Project Size: 175 hours

1. About me

- **Full Name:** Yashaswi Kumar
- **University:** Bangalore Institute Of Technology, INDIA
- **Email:** cestercian@gmail.com
- **GitHub:** cestercian
- **LinkedIn:** cestercian
- **Timezone:** IST (UTC+5:30)
- **Availability (UTC):** 11:30 UTC – 18:30 UTC on weekdays

2. How many hours I can work per week

I will work an average of 28-30 hours every week.

3. Other commitments

1. College Exams
2. Course projects

These are going to be an integral part of my 2nd semester activities. They are not going to affect my GSOC efforts.
I will be able to deliver my committed number of hours per week without much hindrance.

4. Chosen Idea

I wish to contribute to the **OpenWISP netjsongraph.js** project, specifically focusing on the idea:

"Improve netjsongraph.js Resiliency and Functionality".

Abstract

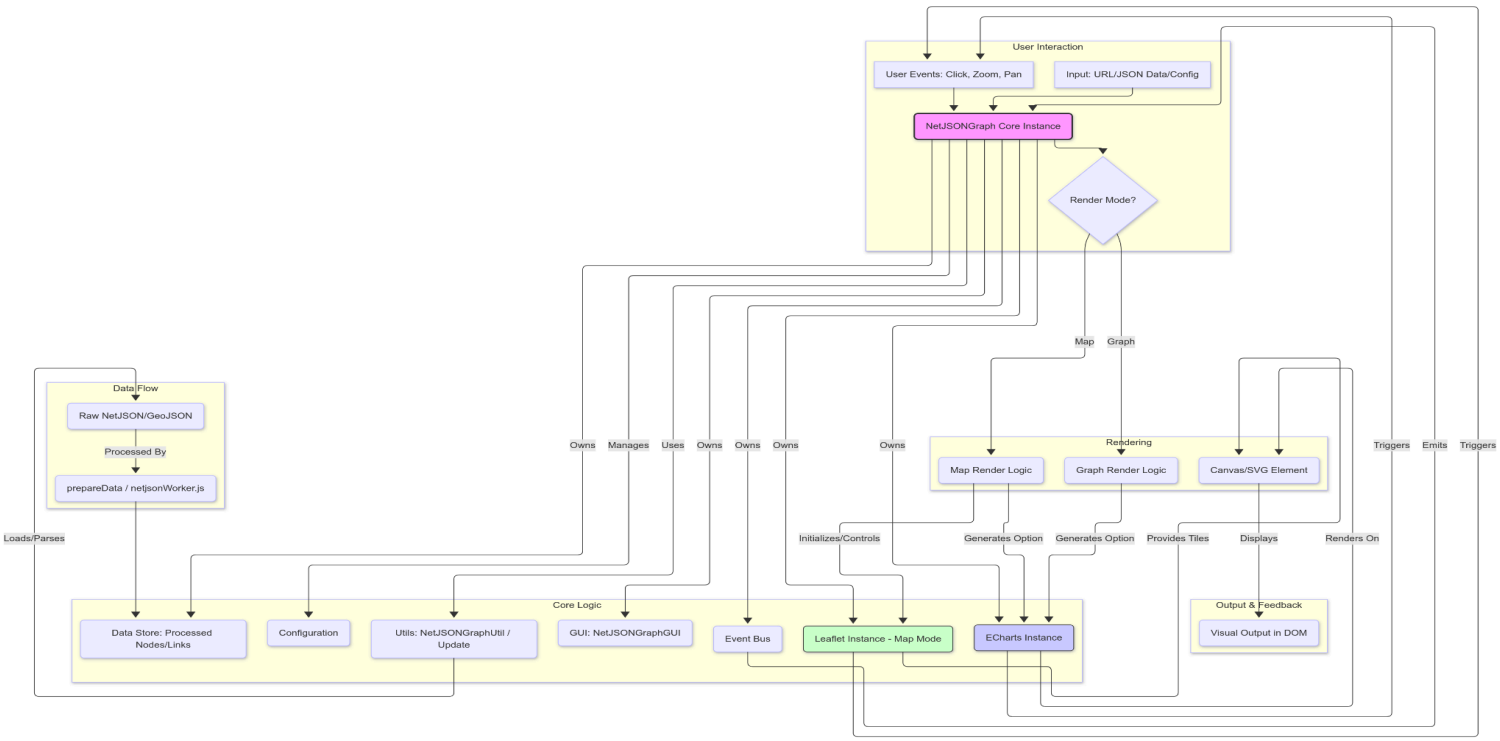
- Show node names on high zoom levels by default
- Respect zoom levels of tile providers; make configurable
- Prevent overlapping of clusters with the same location
- Add resiliency for invalid data; log errors
- Display additional data on nodes (e.g. connected clients)
- Show node labels only after a configurable zoom level

5. Proposal Description

Summary

netjsongraph.js is a JavaScript library designed for visualizing network topology data using the **NetJSON** format. It leverages powerful rendering engines like **ECharts** for both force-directed graph layouts and as an overlay on geographic maps provided by **Leaflet**. This project aims to enhance the library's robustness (**Resiliency**) against invalid data and expand its feature set for improved usability and information density, particularly concerning map **Zoom Levels**, node/link representation, and **Clustering**.

- **NetJSON:** A data interchange format for network topology, defining nodes and links with properties.
- **ECharts:** A comprehensive charting and data visualization library used as the primary rendering engine for graphs and map overlays.
- **Leaflet:** An open-source JavaScript library for interactive maps, used here to provide the base geographic map layer.
- **Tile Provider:** A service supplying map images (tiles) at various zoom levels (e.g., OpenStreetMap, Stadia Maps).
- **Clustering:** Grouping nearby map markers (nodes) into a single cluster icon, often used to manage density at lower zoom levels. leaflet.markercluster is the plugin used.
- **Resiliency:** The library's ability to handle unexpected or invalid input data gracefully (e.g., logging warnings) without crashing.
- **Zoom Level:** The magnification level of the map or graph view.



Goals:

Goal 1/6: Show node labels only after hitting a certain zoom level (Refined based on Issues #148 & #144)

- ***Problem**:* Node labels on geographic maps can clutter the view when zoomed out.

- ***Context** (from GitHub Issue #148):* Discussions revealed an existing config option, `showLabelsAtZoomLevel(default 7)`, for map mode. Mentor `@nemesifier` suggested exploring changing the names of labels and styles to make them easily readable in the graph mode.

- ***Objective**:*
  - **Verify & Refine (Map Mode):** Test the existing showLabelsAtZoomLevel thoroughly. Confirm it's the effective default behavior for hiding labels at low zoom.
  - **Implement (Graph Mode):** Investigate using `ECharts` events (graphroam/datazoom) to implement similar functionality for the graph layout, evaluating performance.
  - **Testing:** Add browser tests for both modes, asserting label visibility changes correctly at the threshold during zoom.
  - **Documentation:** Document the feature clearly for both modes.

- ***Approach**:* Review map code related to showLabelsAtZoomLevel. Test map mode. Investigate ECharts events for graph mode and assess setOption performance impact. Implement graph features if viable. Add tests. Update docs.

- ***Deliverable**:* Verified map label logic, potentially implemented graph label logic, documentation, and tests.

```
// In /src/js/netjsongraph.render.js within graphRender or a related method

// Add listener after echarts init
this.echarts.on('graphRoam', this._handleGraphRoam, this);

// ...

_handleGraphRoam: function() {
    // Debounce this function if performance is an issue
    const currentZoom = this.echarts.getOption().series[0].zoom;
    const threshold = this.config.showLabelsAtZoomLevel || 7; // Use configured
threshold
    const showLabels = currentZoom >= threshold;



    // Only update if the state needs to change
    if (showLabels !== this._currentGraphLabelState) {
        this.echarts.setOption({
            series: [{
                label: {
                    show: showLabels
                }
            }]
        });
        this._currentGraphLabelState = showLabels; // Track state
    }
},
```

Goal 2: Map should respect zoom levels of tile providers (Based on Issue #188)

- **Problem:**.When users zoom into the map beyond the zoom levels that the background tile provider supports, they may encounter areas of the map where no image tiles are available. This results in blank or missing sections of the map, which can create a frustrating and confusing user experience.

- **Context (from GitHub Issue #139):** The initial solution of setting a static maxZoom was rejected because it didn't consider that different tile layers might have different zoom limits. The preferred solution is to dynamically derive the minZoom and maxZoom from the currently active tile layer, with a sensible default as a fallback.

- **My Objective:** My objective for this goal is to modify the map rendering logic to dynamically read the `minZoom` and `maxZoom` values specified within the options

object of the *active* tile layer entry in the `mapTileConfigarray`. These retrieved values will then be used to constrain the Leaflet map instance's zoom capabilities. If the tile provider configuration doesn't specify these limits, sensible defaults (e.g., minZoom: 3, maxZoom: 19) will be applied. I will also ensure that users can still *override* these limits via the main `mapOptions.minZoom` and `mapOptions.maxZoom` configuration if they need specific constraints regardless of the tile provider.

- **My Approach:**
    - **Locate Logic:** The core changes will be within the mapRender function in `/src/js/netjsongraph.render.js`, specifically after the Leaflet map instance `(this.leaflet)` is initialized and the tile layers are added.
    - **Identify Active Tile Layer:** Determine the configuration for the currently active base map tile layer. In the default setup with a single entry in `mapTileConfig`, this is straightforward. If multiple layers are configured (like in netjsonmap-multipleTiles.html), the logic needs to identify the *base layer* currently being displayed.
    - **Read Tile Limits:** Access `this.config.mapTileConfig[activeIndex].options.minZoom` and `this.config.mapTileConfig[activeIndex].options.maxZoom`.
    - **Determine Final Limits:** Establish a precedence order:
    - Use `this.config.mapOptions.minZoom/maxZoom` if explicitly set by the user (highest priority).

    - Otherwise, use the values read from the active tile layer's config in `mapTileConfig`.
    - If still undefined, fall back to sensible defaults (e.g., minZoom: 3, maxZoom: 19 - to be finalized based on common provider limits). These defaults could be defined within `netjsongraph.config.js` or directly in the `mapRender` logic.

- **Apply Limits:** Use the Leaflet map instance's methods: `this.leaflet.setMinZoom(finalMinZoom)` and `this.leaflet.setMaxZoom(finalMaxZoom)`.

- **Testing:** Modify browser tests `test/netjsongraph.browser.test.js` to load maps with tile layers having defined minZoom/maxZoom and assert that the Leaflet instance respects these limits.

- **Documentation:** Update the README.md or relevant documentation to explain this behavior, the configuration precedence, and how to set limits in

mapTileConfig.

- **Deliverable:**
  - Modified mapRender function in `/src/js/netjsongraph.render.js` implementing the dynamic zoom limit logic.
  - Updated default values in `/src/js/netjsongraph.config.js` if necessary.
  - Browser tests verifying the zoom constraints under different configurations (tile-defined, user-override, default).
  - Clear documentation outlining the feature and configuration options.

Goal 3: Prevent Overlapping Clusters and Implement Category-Specific Styling (Refined based on Issue #171 & PR #349):

- **Problem** : When clustering nodes by a clusteringAttribute (e.g., 'status'), clusters representing *different* categories at the exact same map location visually stack on top of each other, making them indistinguishable. Additionally, individual nodes (not part of clusters) are sometimes rendered with a default blue color instead of the color specified in nodeCategories.

- **Context** (**from GitHub Issue #171**) : originally reported the cluster overlap problem. PR #349 introduced improvements to makeCluster using KDBush for better spatial + attribute grouping and added clusterUtils.js with preventClusterOverlap for post-render DOM manipulation to visually separate overlapping markers. The need for category-based styling for both clusters and individual nodes was also highlighted.

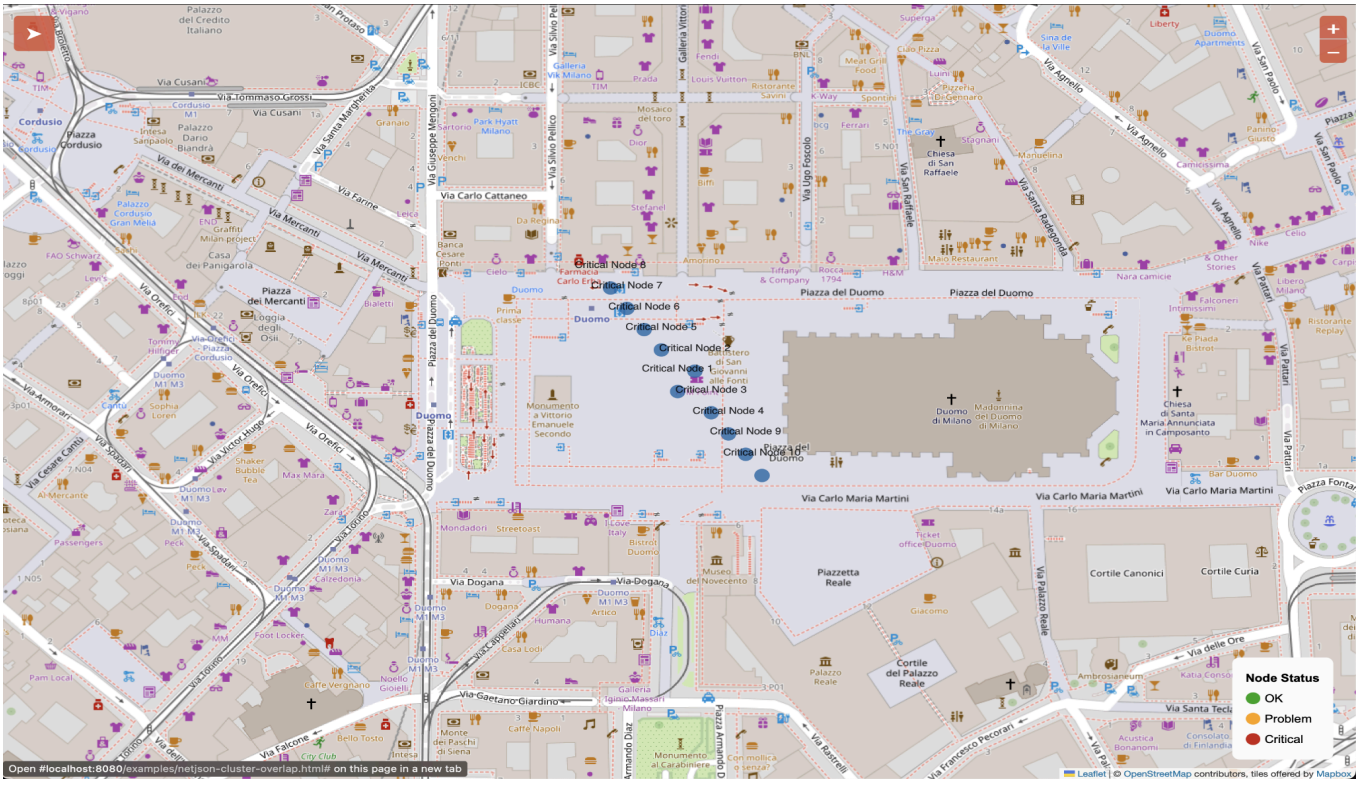
- **Objective** : Finalize and integrate the overlap prevention mechanism from PR #349, ensuring that clusters of different categories at the same location are visually distinct. Implement dynamic, category-specific styling for both Leaflet.markercluster icons and individual map node markers, ensuring they correctly reflect the styles defined in the nodeCategories configuration.

- **Approach**:
  - **Finalize PR #349 Core Logic :** Rebase the PR branch onto the latest master and address any outstanding feedback.

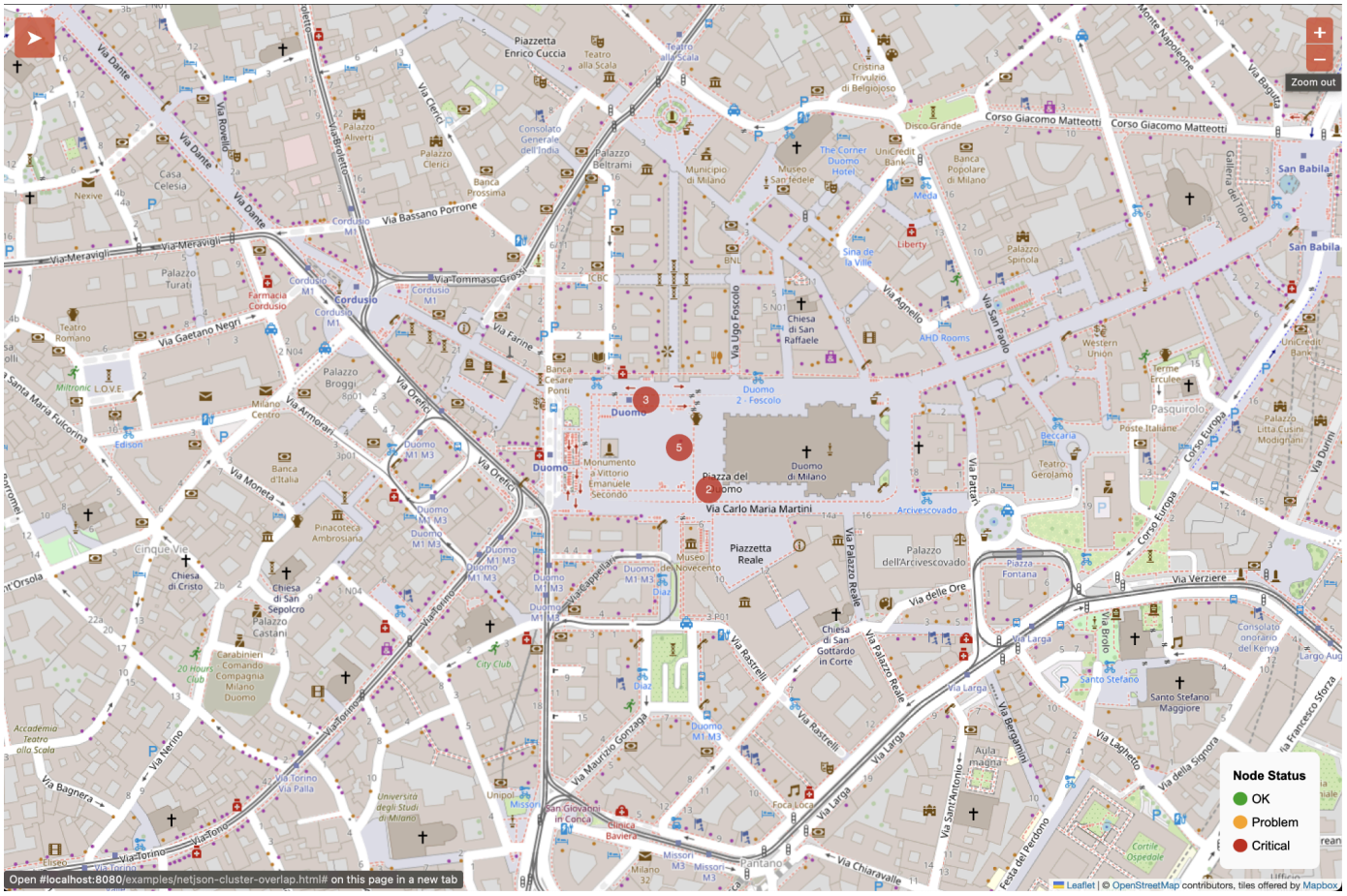
  - **Verify makeCluster (src/js/netjsongraph.util.js):** Confirm it correctly groups nodes by *both* proximity (clusterRadius) and clusteringAttribute. Ensure it assigns necessary category information (e.g., a specific CSS class or style data) to the generated cluster data structures, even for clusters originating from the same coordinates but different categories.

  - **Validate clusterUtils.js:** Confirm the DOM manipulation reliably selects (.marker-cluster) and repositions overlapping clusters using CSS transforms based on getBoundingClientRect. Ensure Leaflet event listeners (zoomend, moveend, etc.) in setupClusterOverlapPrevention correctly trigger the repositioning.

- **Implement Category-Specific Styling:**
  - **Cluster Icon Styling:** Provide a custom iconCreateFunction when initializing L.markerClusterGroup. This function will read the category information (e.g., the categoryClass added in makeCluster) and apply it to the cluster icon.



When Zoomed In



When Zoomed Out

- **Individual Node Styling:** Modify the node marker creation logic (likely in _renderMap or a helper function) to correctly apply styles from nodeCategories.

- **Testing**:
  - **Unit Tests:** Enhance tests for makeCluster to assert correct grouping by attribute and assignment of category information (class/style) to cluster data.
  - **Integration/Browser Tests (test/netjsongraph.browser.test.js):**
  - Use netjson-cluster-overlap.html or similar examples.
  - **Overlap:** Verify that clusters with different categories at the same initial coordinates have different transform styles applied after preventClusterOverlap runs (e.g., upon load, zoom, pan). Check their screen positions are separated.
  - **Cluster Styling:** Inspect the generated cluster divIcon elements. Assert they have the correct category-specific CSS class (e.g., cluster-category-ok) and that their computed background-color matches the expected category color.
  - **Node Styling:** Inspect individual L.circleMarker elements (or their SVG/Canvas paths). Assert their fill and stroke attributes match the colors defined in nodeCategories.
  - Test scenarios with only one category cluster, multiple different category clusters overlapping, and mixtures of clusters and individual nodes.

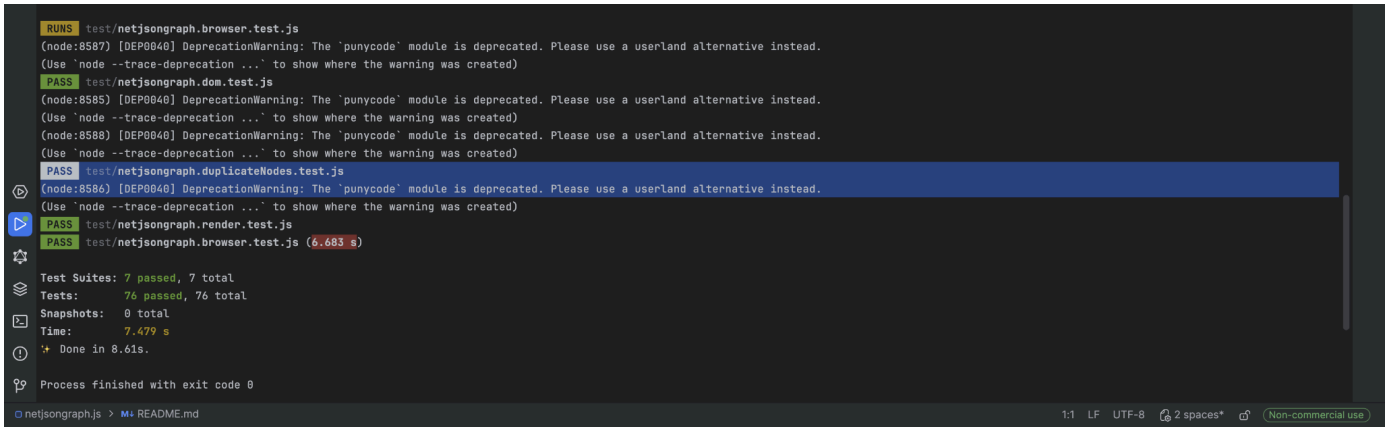
- **Documentation**:
  - **Update README.md**
  - Refine the explanation of clusteringAttribute to clarify it works in conjunction with spatial proximity.
  - Explain the two-part overlap solution: logical separation by makeCluster and visual separation by clusterUtils.js.
  - Document how cluster icon and individual node colors are now derived from nodeCategories.
  - Ensure the netjson-cluster-overlap.html example clearly demonstrates both features and includes comments explaining the relevant configuration (clusteringAttribute, nodeCategories).

- **Deliverable:**
  - A merged Pull Request (likely updating #349) containing:
  - Finalized makeCluster logic with attribute grouping and category data propagation.
  - Integrated clusterUtils.js for visual overlap prevention.
  - Custom iconCreateFunction for category-styled cluster icons.
  - Corrected category styling logic for individual node markers.

- Comprehensive unit and browser tests covering overlap and styling.
- Updated README.md and examples.

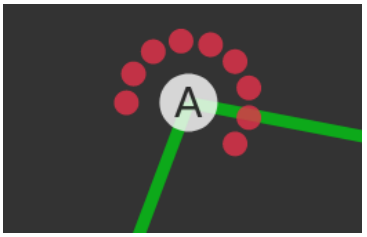
Goal 4: Add Resiliency for Invalid Data (Refined based on Issue #164 & PR #355):

- **High Level Problem:** Providing NetJSON data with duplicate node IDs or links pointing to non-existent nodes can cause the library to crash.
- **Low Level Context:** Issue #164 reported the duplicate ID crash. My PR #355 centralizes duplicate node removal using deduplicateNodesById in `/src/js/netjsonWorker.js`, called by `dealJSONData`, `addData`, and `mergeData`. Test effectiveness needs final verification. Invalid link handling is not yet implemented.
- **My Objective:** 1. Finalize PR #355, ensuring tests robustly verify the crash prevention for duplicate nodes. 2. Implement checks to skip rendering links with invalid source or target IDs, logging warnings.
- **My Approach:**
  - **PR #355:** Finalize code, address feedback. Crucially, adapt tests in /test/netjsongraph.duplicateNodes.test.js to replicate the *update sequence* that caused the original crash on master, then confirm they pass with the fix. Verify console.warn calls.
  - **Testing:** Added Jest tests for invalid link scenarios. Update browser tests if needed.
  - **Documentation:** Update README regarding improved resilience and console warnings.
  - **Deliverable:** Merged solution for duplicate IDs. Added handling for invalid links. Console warnings. Comprehensive tests confirming crash prevention. Updated docs.
- **Deliverable:**
  - A merged pull request (based on #355) with finalized, robust, and verified handling of duplicate node IDs using a centralized utility, preventing related crashes.



- Implementation within rendering functions to check for and skip links with non-existent source or target nodes.
- Clear console.warn messages indicating skipped duplicate nodes and invalid links.
- Comprehensive unit and potentially browser tests demonstrating that the library no longer crashes and logs appropriate warnings for both duplicate nodes and invalid links during initial load and updates.
- Updated documentation reflecting the library's enhanced data handling resilience.

- **Goal 5: Display Additional Data (e.g., Connected Clients) on Nodes (Refined based on Issue #153):**
  - **Problem:** Users need a way to visualize secondary numeric data associated with nodes (like client counts shown in Meshviewer) directly on the graph/map.



  - **Context (from GitHub Issue #153):** Issue #153 requested this feature. Concept code using canvas drawing was provided by @piy3.
  - **My Objective:** Implement a configurable feature using nodeVisualAttributes option to display specified numeric node properties as visual indicators (dots/badges) near nodes in both graph and map modes.
  - **My Approach:**
    - **Configuration Design:** I will introduce a new configuration array, potentially named nodeVisualAttributes, within the main library options. Each object in this array will define a visual indicator:

```
nodeVisualAttributes: [
  {
    property: 'clients_wifi24', // Key in node.properties or node root
    style: { shape: 'circle', color: '#ff0000', size: 4 }, // Style for the
indicator
    position: 'top-right', // Predefined relative position (e.g., 'top-right',
'bottom-left', or potentially orbital like Meshviewer if feasible later)
    // Optional: badge: { show: true, formatter: '{value}' } // For displaying
the number
  },
  {
    property: 'clients_wifi5',
    style: { shape: 'circle', color: '#0000ff', size: 4 },
    position: 'bottom-right',
  },// Add more attributes as needed]
```

- **Implementation Strategy (ECharts Graphic Components):** I will primarily leverage ECharts' graphic component system. This seems the most versatile approach applicable to both graph (graph/graphGL) and map (scatter/effectScatter on Leaflet) modes.
    - **Data Iteration:** During the data processing phase (likely within generateGraphOption and generateMapOption in /src/js/netjsongraph.render.js where the series.data is prepared), I will iterate through each node *after* its primary layout position is determined.
    - **Indicator Generation:** For each node and each entry in nodeVisualAttributes, I will:
        - Read the numeric value of the specified property from the node's data (node.properties[indicatorConfig.property] or node[indicatorConfig.property]).
        - If the value exists and is numeric, generate an ECharts graphic element object (e.g., type 'circle', 'rect', or potentially 'text' for badges).
        - Find the x, y position for the graphic element based on the node's position and position configuration. Start with simpler relative positioning instead of orbital placement.
        - Apply the configured style (color, size) to the graphic element.
    - **Adding to Option:** Collect all generated graphic element objects and add them to the echartsOption.graphic array before calling echarts.setOption. ECharts will handle rendering these elements relative to the chart coordinate system.
- **Handling Updates:** Ensure that when setOption is called for data updates, the graphic elements are appropriately updated, added, or removed. This might involve assigning unique IDs or names to the graphic elements based on the node ID and attribute property.
- **Flexibility:** The design allows users to map *any* numeric property to a visual indicator, not just hardcoded client counts. Styling and positioning are configurable per attribute.
- **Testing:**
    - Add unit tests for the configuration parsing and graphic element generation logic.
    - Create browser tests using datasets containing the configured properties (e.g., mock client counts) to visually verify:
        - Indicators appear correctly in both graph and map modes.
        - Indicators are positioned according to the position setting.
        - Styles (color, size) are applied correctly.
        - Indicators update correctly when data changes via JSONDataUpdate.
- **Documentation:** Update the README.md with a new section detailing the nodeVisualAttributesconfiguration option, its parameters (property, style, position, badge), and provide clear examples.

- **Deliverable:**
  - A new configuration option (nodeVisualAttributes) allowing users to specify node properties for visual indication.
  - Implementation using ECharts graphic components to render these indicators in both graph and map modes.
  - Clear documentation explaining the configuration and usage.
  - Examples demonstrating how to visualize different numeric properties (like client counts) using this feature.
  - Unit and browser tests verifying the functionality and visual output.

```
// In /src/js/netjsongraph.render.js -> generateGraphOption /
generateMapOption

const nodeVisualAttributesConfig = this.config.nodeVisualAttributes || [];
const graphicElements = [];

processedNodes.forEach((node, dataIndex) => { // ... existing node
processing ...

const nodeLayout = node; // Or use layout from
data.getItemLayout(dataIndex)

const nodeSize = node.symbolSize; // Get final node size

nodeVisualAttributesConfig.forEach((attrConfig, attrIndex) => {

    const value = node.properties?.[attrConfig.property] ??
node[attrConfig.property];

    if (value != null && !isNaN(value)) {

        // Calculate position based on nodeLayout.x, nodeLayout.y,
nodeSize, and attrConfig.position

        const indicatorPos = calculateIndicatorPosition(nodeLayout,
nodeSize, attrConfig.position, attrIndex); // Implement this helper

        graphicElements.push({

            type: attrConfig.style?.shape || 'circle', // Default to circle

            id: `nodeAttr_${node.id}_${attrConfig.property}`, // Unique ID
            x: indicatorPos.x,
            y: indicatorPos.y,
            z: node.z + 1, // Ensure it's slightly above the node

            style: {

                fill: attrConfig.style?.color || '#ff0000',

                // ... other style properties ...

            },
```

```
            shape: { // Shape specific properties

                r: attrConfig.style?.size || 4, // Example for circle
                // width/height for rect etc.

            },

            // Potentially add badge text element here too if configured

        });
    }
});


// Add to the final ECharts option echartsOption.graphic = { elements:
graphicElements };
```

Testing and Documentation:

Throughout the project, I will ensure that:

6. Existing tests are maintained and passed.
7. New tests (using Jest) are added to cover the new features and resiliency improvements. Browser tests (using Selenium/WebDriver) will be added/updated for visual and interaction changes.
8. The README.md and any relevant documentation are updated to reflect the changes and new features, including clear explanations and examples for the new configuration options..

---

7. Implementation Plan & Detailed Timeline (175 hours)

(Approx. 15 hours/week)

Phase 0: Community Bonding (Weeks 1-2 - Est: 15 hrs total)

- ***Week 1:***
    - *Finalize development environment setup (Node.js, npm/yarn, Git, IDE).*
    - *Perform a deep dive into the netjsongraph.js codebase, focusing on:*
        - *Core structure (netjsongraph.core.js, netjsongraph.js).*
        - *Rendering logic (netjsongraph.render.js - specifically mapRender and graphRender).*
        - *Configuration (netjsongraph.config.js).*
        - *Utilities (netjsongraph.util.js, netjsongraph.update.js).*
        - *Data processing worker (netjsonWorker.js).*
        - *Existing tests (test/ directory).*
    - *Clone repository, ensure all existing tests pass locally.*
    - *Initial meeting with mentors (Nitesh Sinha, Federico Capoano) to confirm understanding, communication plan, and refine weekly goals.*

- ***Week 2:***
    - *Continue code exploration, focusing on Leaflet integration (echarts-leaflet/index.js) and the leaflet.markercluster plugin usage.*
    - *Analyze the existing showLabelsAtZoomLevel implementation for map mode.*
    - *Begin verifying Goal 1 (Map Label Zoom): Manually test the default label behavior in `netjsonmap.html` across different zoom levels.*
    - *Review PR #349 (Clustering) and PR #355 (Duplicate Nodes) code and feedback.*

Phase 1: Zoom Features & Initial Resiliency (Weeks 3-5 - Est: 45 hrs)

- ***Week 3:***
    - ***Goal 1 (Map Labels):*** *Complete verification. Identify any bugs or edge cases. Ensure it's the default behavior. Start documenting showLabelsAtZoomLevel for map mode. Add initial Jest/browser tests for map label visibility.*
    - ***Goal 2 (Tile Limits):*** *Begin implementation in mapRender. Add logic to read minZoom/maxZoom from active mapTileConfig entry. Implement fallback defaults.*
    - *Communication: Weekly update, discuss any findings on Goal 1.*

- ***Week 4:***
    - ***Goal 2 (Tile Limits):*** *Finish implementation, including handling user overrides from mapOptions. Add Jest/browser tests to verify zoom limits are correctly applied. Document the feature and precedence.*
    - ***Goal 1 (Graph Labels):*** *Start investigation for graph mode. Add graphRoam event listener in graphRender. Implement logic to get zoom level and conditionally call*

echarts.setOption to toggle series.label.show.
    - *Communication: Weekly update, share progress on Goal 2 and initial findings for Goal 1 (Graph).*

- ***Week 5:***
    - ***Goal 1 (Graph Labels):*** *Assess performance impact of setOption during graph roam, especially with large datasets (e.g., netjsongraph-graphGL.json). Implement debouncing if necessary or decide on feasibility with mentors. Finalize implementation or document limitations. Add tests if implemented. Update documentation for both modes.*
    - ***Goal 4 (Resiliency - Duplicates):*** *Begin finalizing PR #355. Rebase onto master. Address any outstanding comments.*
    - *Communication: Weekly update, crucial discussion on graph label performance.*

Phase 2: Resiliency & New Functionality (Weeks 6-8 - Est: 45 hrs)

- ***Week 6:***
    - ***Goal 4 (Resiliency - Duplicates):*** *Refine tests for PR #355 to ensure they effectively reproduce the original crash scenario on master and pass with the fix. Ensure console.warn works. Aim to get PR ready for final review/merge.*
    - ***Goal 4 (Resiliency - Links):*** *Implement checks in generateGraphOption and generateMapOption to skip links with non-existent source or target nodes, logging warnings.*
    - *Communication: Weekly update, focus on test validation for Goal 4.*
- ***Week 7:***
    - ***Goal 4 (Resiliency - Links):*** *Add comprehensive Jest tests for invalid link handling scenarios. Update documentation regarding overall resiliency improvements.*
    - ***Goal 5 (Node Visual Attrs):*** *Design the nodeVisualAttributes configuration structure. Start implementing the logic to read config and generate ECharts graphic element objects for graph mode within generateGraphOption.*
    - *Communication: Weekly update, confirm Goal 4 completion, discuss Goal 5 config design.*
- ***Week 8:***
    - ***Goal 5 (Node Visual Attrs):*** *Continue implementation. Adapt logic for map mode (generateMapOption), ensuring graphic elements work correctly with Leaflet overlay. Implement basic positioning (e.g., top-right offset).*
    - *Communication: Weekly update, demonstrate initial visual indicators.*

Phase 3: Clustering & Midterm (Weeks 9-10 - Est: 30 hrs)

- ***Week 9:***
    - ***Goal 5 (Node Visual Attrs):*** *Refine implementation, add styling options based on config. Add initial tests and examples.*

    - ***Goal 3 (Clustering Overlap/PR #349):*** *Rebase PR #349 again if needed. Address any remaining feedback. Focus on testing the overlap prevention logic (both makeCluster and clusterUtils.js if applicable).*
    - *Communication: Weekly update, show progress on Goal 5, discuss approach finalization for Goal 3.*
- ***Week 10:***
    - ***Goal 3 (Clustering Styling):*** *Implement dynamic cluster icon coloring based on contained node categories within the iconCreateFunction. Handle single vs. multiple category cases. Add tests.*
    - ***Documentation & Code Review Prep:*** *Update documentation for Goals 1, 2, 4, 5. Ensure code for completed goals is clean and well-commented.*
    - ***Midterm Evaluation:*** *Prepare and submit evaluation materials. Discuss progress thoroughly with mentors.*
    - *Communication: Midterm Check-in meeting.*

Phase 4: Integration, Testing, Finalization (Weeks 11-13 - Est: 40 hrs + Buffer)

- ***Week 11:***
    - ***Goal 3 (Clustering):*** *Finalize testing (overlap prevention, styling, event handling). Merge PR #349 after final approval. Update documentation for clustering.*
    - ***Comprehensive Testing:*** *Begin end-to-end testing of all new features and fixes across different browsers (Chrome, Firefox) and modes (graph/map). Use various example files.*
    - *Communication: Weekly update, highlight testing progress and any discovered bugs.*

- ***Week 12:***
    - ***Bug Fixing:*** *Address any issues found during comprehensive testing or from final mentor feedback.*
    - ***Code Cleanup & Refactoring:*** *Ensure code consistency, remove dead code, add necessary comments.*
    - ***Documentation Finalization:*** *Complete all README updates, configuration explanations, and finalize examples demonstrating new features.*
    - *Communication: Weekly update, focus on bug fixes and documentation status.*

- ***Week 13 (Final Week / Buffer):***
    - ***Final Testing:*** *Perform final regression tests.*
    - ***Prepare Submission:*** *Create final pull requests for any outstanding work. Ensure all code meets contribution guidelines. Prepare the final GSoC project submission.*
    - ***Presentation Prep:*** *(If required) Prepare a brief presentation or demo of the completed work.*
    - *Communication: Final check-in with mentors, confirm submission details*

---

8. Why NetJSONGraph.js?

The opportunity to contribute to the netjsongraph.js project is very exciting! My strong JavaScript skills and experience with ECharts align perfectly with the project's visualization goals. I'm particularly interested in the challenge of improving the robustness and user-friendliness of libraries – it's work that I find both engaging and rewarding.

The prospect of working on both force-directed graphs and Leaflet map rendering within the same project is especially appealing. Furthermore, I'm eager to contribute to OpenWISP due to my strong belief in the value of open-source networking tools. I'm excited by the possibility of helping to enhance netjsongraph.js, making it even more reliable and feature-rich for users who are working with complex network visualizations.

---

9. Past Contributions and Communication

Open Source Contributions:

I've been actively contributing to openwisp/netjsongraph.js recently, demonstrating my commitment to this project:

📁 Repo: netjson/netjsongraph.js

S.No.	PR No.	Description	Status
1	#355	[fix] Fixed duplicate node ID handling	Review required
2	#354	[Fix] disableClusteringAtLevel not applied on initial render	Review required
3	#349	[Fix] Prevent Overlapping of Clusters in netjsongraph.js	Changes requested
4	#344	[chores] Refactor CSS for consistency and clarity	Merged

---

📁 Repo: google-gemini/api-examples

S.No.	PR No.	Description	Status
5	#38	Added Go implementation of Files API with test suite	Review required
6	#37	Added count_tokens implementation for Go	Review required
7	#36	Added Go Cache Implementation and Tests with Documentation	Review required
8	#31	Implemented JavaScript example for configuring Gemini model parameters	Review required

---

📁 Repo: google-gemini/gemini-image-editing-nextjs-quickstart

S.No.	PR No.	Description	Status
9	#16	Improve API Error Handling and Response Format	Review required
10	#15	Fix image component styling and form handling	Review required
11	#13	Enhance ImageUpload Component with Error Handling, Loading State, and A11y	Merged

---

📁 Repo: google-gemini/generative-ai-js

S.No.	PR No.	Description	Status
12	#401	Enhance generateContentStream with streamCallbacks support. Fixes #322	Review required

---

📁 Repo: juspay/hyperswitch

S.No.	PR No.	Description	Status
13	#7659	Improve (Payment Templates): Complete, Default, and Resolve Variables	Review required

---

📁 Repo: zendalona/zendalona

S.No.	PR No.	Description	Status
14	#2	Improve accessibility and code readability in UI elements	Open

This history reflects my ability to dive into different codebases, fix bugs, add features, improve code quality, and collaborate effectively within open-source communities.

Relevant Project Experience:

My personal projects further highlight my practical experience with JavaScript, React, and building web applications:

- **Coder :** A JavaScript web app for anonymous code storage and sharing, focusing on front-end logic and user interaction.

- **Recipe App:** A React application utilizing external APIs to fetch and display data, showcasing skills in React, state management, and asynchronous JavaScript. ([Link to cestercian/Recipe repo])
- **Hangman Game :** A React-based game demonstrating component structure and UI logic.

These projects, viewable on my GitHub, demonstrate my core skills in JavaScript, React, HTML/CSS, and interacting with APIs – all relevant to enhancing netjsongraph.js.

Communication:

I plan to maintain clear, consistent, and proactive communication:

- **Regular Updates:** I will provide weekly summaries of progress, challenges, and next steps, likely via GitHub issues/PRs and in a dedicated development and community support channel in the element app as preferred by the mentors.
- **Mentor Check-ins:** I am available and committed to attending regular meetings with mentors, respecting time zones, to discuss progress, seek guidance, and align on priorities.
- **Proactive Questions:** I will ask questions promptly when I encounter blockers or need clarification, using the agreed-upon communication channels (GitHub preferred for technical discussions).

Learning Ability:

When encountering new technologies or unfamiliar code sections, my approach is to first consult documentation and relevant examples within the project. I break down problems into smaller parts and experiment. If I remain stuck after initial research, I focus on formulating clear, specific questions for mentors or the community. I view challenges as learning opportunities and am eager to deepen my understanding of ECharts, Leaflet, and building robust visualization libraries through this project.

8. **After GSoC**

- **Continued Collaboration:** Yes, I am definitely interested in continuing to collaborate with OpenWISP after GSoC. My motivation stems from my positive experience with the community, my interest in the project domain, and the desire to see the features I implement further refined and adopted. Contributing long-term allows for deeper learning and a greater sense of ownership.
- **Maintenance:** Yes, I am willing to help maintain the features I implement for a reasonable period after GSoC, addressing bugs and potentially assisting with minor enhancements as my availability permits. I understand the importance of maintaining contributions in open source.

OpenWrt Experience

- **Experience with OpenWrt:** I currently have limited direct experience with OpenWrt firmware itself.
- **Router Availability:** Yes, I have a router at home ([Specify Router Model if known, e.g., TP-Link Archer C6]) on which I can flash OpenWrt for testing purposes related to OpenWISP integration, if necessary for context, although this specific project focuses on the JS library.

Freelance Opportunities: Yes, I would be interested in exploring occasional freelance paid work for OpenWISP if opportunities arise that align with my skills and availability. My primary motivation remains learning and contributing to impactful open-source software.